A Computer Music Implementation Course Using Active Essays

Mark Guzdial and Jim Greenlee
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
guzdial@cc.gatech.edu,jkg@cc.gatech.edu

Abstract

Computer music tends to be the domain of musicians, electrical engineers, mathematicians, and physicists. Today, only CS1-level knowledge is necessary to do a serious review and exploration of computer music, as the algorithms have become more well understood (and thus easier to explain), and Moore's Law makes even inefficient algorithms successful. A serious impediment, however, has been the lack of a text that is written from a programming perspective (instead of mathematical or musical). By using active essays in Squeak, a set of notes have been created where students can read the code, execute it, hear the result, see the waveform, and take the Fourier transform.

1 The Domain of Computer Music

Computer music is an unusual area of study in that the word "computer" is in the name, yet few Computer Science departments offer classes in it. Instead, computer music is often taught as part of digital signal processing courses in Electrical Engineering; perhaps as a special topic in music course; and perhaps as part of a physics course on the *Physics of Music*.

There are several reasons for considering incorporating computer music into Computer Science curricula:

- Music on computers is a growing common concern (rf. Napster), and appears across the Web in various guises. With such a pervasive medium, computer professionals will doubtlessly need to work with it. Knowledge of the variables and algorithms of computer music would better prepare our students for their careers.
- Recent reports exploring why students drop-out (or completely avoid) computer courses suggest that part of the problem is simply the tedious, overly-technical assignments that we require which involve very little creativity [1]. Computer music offers the opportunity to teach about computer science using assignments that might engage a different kind of student and that do allow for a good bit of creativity.
- While other departments may teach the underlying physics and mathematics of music and sound, or may teach how to use computer music applications, the implementation of the applications and the underlying algorithms for generating music is the domain of Computer Science.

Existing computer music texts generally do not emphasize how to implement the underlying algorithms, which poses a problem for the Computer Science teacher who wishes to teach this content. There *are* many excellent books on computer music. Curtis Roads' book [7] is a comprehensive tutorial over all of computer music, but with little on implementation. The new book on the popular music programming language *Csound* by Richard Boulanger [2] provides many examples of Csound programs (instrument definitions), but with little explanation of how the oscillators, resonators, and filters are implemented. Our favorite book for describing the algorithms [3] still does not provide examples of code.

The most common computer music algorithms are not actually difficult to implement. The algorithms for oscillators, filters, and envelope generators are rather simple — they require no more than single (unnested loops) and fewer than a dozen lines of code. In this sense, they are well within the scope of a CS1/CS2 course. Of course, the simplest and easiest to understand implementations are not the most efficient. Moore's Law enables even inefficient algorithms to generate reasonable sound samples in less than 30 seconds. Students

can build interesting sound samples and explore new ones quickly enough to remain motivated.

This paper describes a seven week course taught at Georgia Institute of Technology (Georgia Tech) by the authors, as a portion of a semester long sequence on *Computer Music Implementation*. While the overall course dealt with a wide range of issues in computer music (e.g., composition, improvisation, styles of music), the first half mini-course was focused just on the creation of tones in specific timbres. Very little prior knowledge of music was needed and only CS1-level programming knowledge was required. Thus, this short course is should be easily generalized to other schools in other kinds of curricula. A set of innovative multimedia notes were created to allow students to read, see, hear, and manipulate the example waveforms from within a standard browser.

2 Structure of Course

The course was taught in the Fall 2001 semester as an elective undergraduate course. 20 students enrolled. Lecture and discussions took place in a room equipped with a computer and projector, but without adequate speakers. Instead, we hauled in a laptop computer and speakers for the course.

The textbook for the course was Boulanger's *CSound* book [2]. While the CSound book didn't cover the algorithms we wanted to teach in sufficient detail, it did provide the next level up. CSound is a music language made up of *unit generators* that encapsulate many of the basic algorithms of computer music. CSound became the testing environment for the students—once they got something working in CSound, they could use that as a reference as they debugged their own code.

The course was taught in Squeak [6], a cross-platform programming language based in Smalltalk which has excellent support for multimedia. A pre-requisite for the course was a required course using Squeak [5], but it would probably be possible to teach the course with students learning Squeak at the same time [4].

Squeak has built-in support for generating specific notes ((FMSound pitch: 100 dur: 0.5 loudness: 0.8) play. or (FMSound pitch: 'c3' dur: 0.5 loudness: 0.8) play.), playing with specific timbres, or playing larger scale elements like scales (`(FMSound lowMajorScaleOn: FMSound brass2) play`), so its easy to do the musical equivalent of "Hello, World!" More important, though, is the support to build sounds from the digital ground up. Squeak allows the user to fill a sound buffer with raw integers and then send this sound buffer to a digital-to-analog converter for generating instruments. The course was structured to follow the historical progression from the earliest synthesis algorithms to the latest.

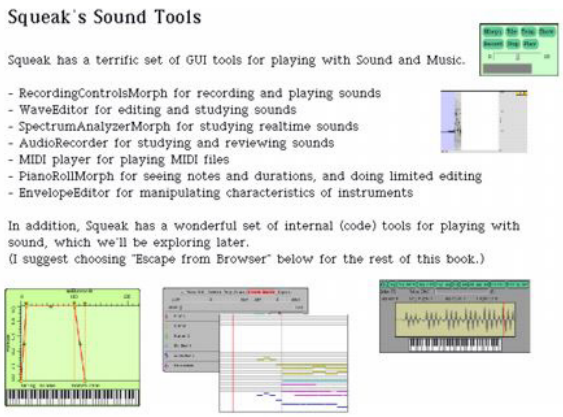

Figure 1: Introductory Lecture to Squeak's Sound Tools

The flow of lectures was:

1. The first lecture introduced the tools that Squeak provides for studying sounds and music. Squeak supports recording sound, displaying waveforms, computing and displaying Fourier transforms, as well as displaying real-time Fast Fourier Transforms (FFTs) and Sonograms (Figure 1).

2. An introduction to acoustics and psychoacoustics explored the physical characteristics of sound (frequency, amplitude) and the relationship to the human perception of this sound. We did several in-class experiments, such as listening to tones very near one another in frequency (e.g., 100 Hz and 101 Hz, then 2000 Hz and 2001 Hz) to see when it was possible to notice a difference—and how that just noticeable difference became larger with higher frequencies.

3. We then did our first synthesis algorithm, additive synthesis where sine waves are simply added together. While additive synthesis generates unusual sounds, it is computationally intense and does not generate the richness of sounds like those found in natural instruments.

4. We then studied the sounds of natural instruments, using a tool called *AudioExplorer* (by Ph.D. student Jochen "Je77" Rick) which allows us to capture FFT in discrete slices. In this way, we were able to see how the spectral content of real instruments changed over time.

5. Next, we implemented an *oscillator* which could play any waveform at any desired frequency. This is a critical component (*unit generator*) used in most computer music synthesis systems.

6. We looked briefly at subtractive synthesis, which requires fewer computational resources than additive

synthesis, but does require *filters*. We created a few digital filters using very simple algorithms.

7. We then introduced Frequency Modulation (FM) synthesis using our oscillator. FM Synthesis was the technique used in Yamaha's popular synthesizers of the 70's. Squeak also includes an implementation of FM synthesis, so two different implementations are compared.

8. Finally, we introduce synthesis based on sampling, which is the technique that most modern synthesizers use today.

3 An Example Lesson

The above list may seem daunting to those unfamiliar with the vocabular, but a brief example will demonstrate how approachable this material is. Additive synthesis, as mentioned, is simply the addition of sine waves at different frequencies. We can implement additive synthesis in about 16 lines of code in Squeak, with nothing particularly sophisticated.

The first method needed is one that creates a sound buffer (in Squeak, an instance of the class SoundBuffer) filled with the appropriate values to generate the desired frequency for the desired duration. The problem requires some trigonometry, but nothing too complex. We need a certain number of values per second, called the *sampling rate*. We know that if we take the sine of the radians from 0 to 2π, we will generate a cycle of a sine wave. If we want 440 Hz as our desired frequency (f), for example, we need to go from 0 to 2π 440 times within the samples defined by the sampling rate (sr). Thus, we need to go from 0 to 2π in $\frac{1}{f}$, and we have $\frac{sr}{f}$ values to do it in.

The actual implementation is only 10 lines of Squeak code.

```
forFreq: freq amplitude: amp duration: seconds
    "Generate a monophonic SoundBuffer filled with a
    sine wave of the given frequency, maximum amplitude,
    and duration in seconds"

    | sr anArray pi interval samplesPerCycle
       maxCycle rawSample |
    sr ← SoundPlayer samplingRate.
    anArray ← SoundBuffer newMonoSampleCount:
       (sr * seconds) .
    pi ← Float pi.
    interval ← 1 / freq. "Time between cycles,
       inverse of frequency: seconds per cycle"
    samplesPerCycle ← interval * sr. "secs/cycle *
        samples/second = samples per cycle"
    maxCycle ← 2 * pi.
    1 to: (sr * seconds ) do: [:sampleIndex |
       rawSample ← ((sampleIndex / samplesPerCycle)
          * maxCycle) sin.
       anArray at: sampleIndex
            put: (rawSample * amp ) rounded.].
    ↑ anArray
```

To add sine waves is pretty complicated in Physics, and involves a lot of math in Electrical Engineering. But in Computer Science, with waves in arrays (sound buffers), the solution is easy: Simply add the samples at the same indices in the buffers! Another six lines of code is needed, with only a single loop.

```
combine: soundbuffer1 and: soundbuffer2
    | newsound |
    (soundbuffer1 size) = (soundbuffer2 size)
       ifFalse:
       [↑self error: 'Sound buffers must be of the
            same length'].

    newsound ← SoundBuffer newMonoSampleCount:
       (soundbuffer1 size).
    1 to: (soundbuffer1 size) do: [:index |
       "Add up each of the samples"
       newsound at: index put:
          (soundbuffer1 at: index) +
             (soundbuffer2 at: index)].
    ↑newsound.
```

We now have enough code to build a waveform. We simply create a few sine waves, add them together, then place them into an object called a SampledSound for playing. The below code can be typed into a Squeak workspace and executed, modified, and executed again. In this way, students can generate a variety of waveforms. The message viewSamples to a sound opens a WaveEditor in Squeak that allows viewing the waveform, editing the wave, taking its Fourier transform, and playing the wave (at various frequencies).

```
|base1 base3 base5 base13 base135 combinedSound |
base1 ← SoundBufferGenerator
   forFreq: 440 amplitude: 4000 duration: 2.
base3 ← SoundBufferGenerator
   forFreq: 3*440 amplitude: 2000 duration: 2.
base5 ← SoundBufferGenerator
   forFreq: 5*440 amplitude: 1000 duration: 2.

base13 ← SoundBufferGenerator combine: base1 and: base3.
base135 ← SoundBufferGenerator combine: base13 and: base5.
combinedSound ← SampledSound samples: base135
      samplingRate: SoundPlayer samplingRate.
combinedSound viewSamples.
"combinedSound play."
```

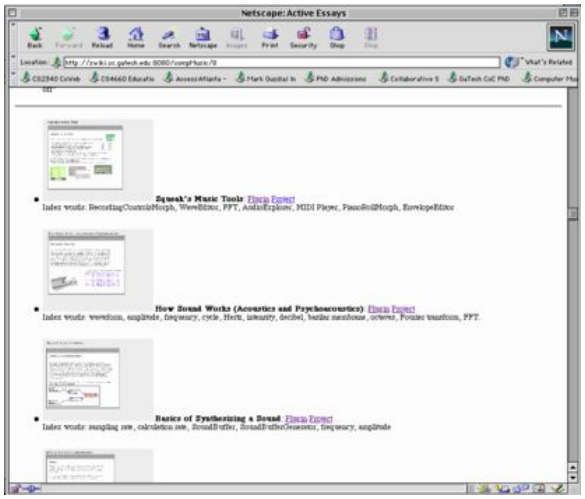

Figure 2: Active Essays List in Web Browser

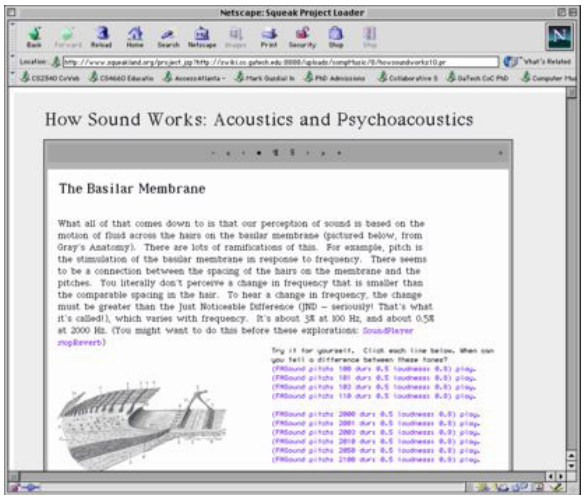

Figure 3: Essay running in Web Browser

4 Use of Active Essays

As mentioned, we were unable to find a textbook that covered exactly the kind of material we wanted to discuss. We developed a set of course notes in a format in Squeak called *Active Essays* developed by Ted Kaehler and Alan Kay [6]. Active Essays are a collection of pages in an electronic book where each page can contain live code that can be executed with a single click or that can be modified for exploration. The Active Essays can do anything that Squeak can do, so we can play sounds, record sounds, run analyses, and see the waveforms from within the Essay. Best yet, Squeak can run as a plugin from within most browsers, so students (and even the instructor in lecture) can run the essay from within a standard Web browser.

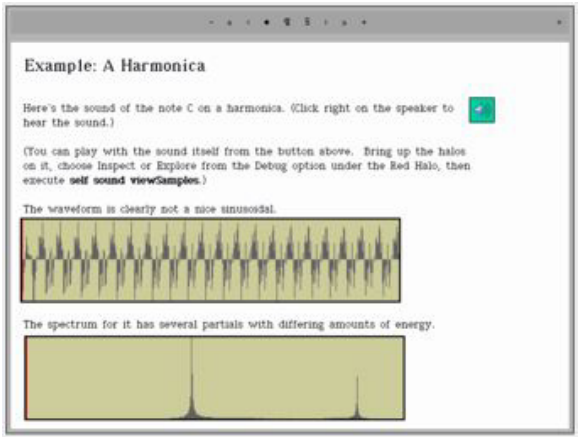

Figure 4: A sound icon, waveform, and Fourier transform in an Active Essay

A half dozen Active Essays were developed for the course. They were made available to the students from a Web page (Figure 2). The students would load their browsers with the Squeak plugin from `http://www.squeakland.org`. When they clicked on the **plugin** link for the Essay they wanted, the essay would open inside the browser (Figure 3).

From within the browser, students could play sounds, see waveforms, and do Fourier transforms of their waveforms (Figure 4). It was also possible to save results from one page and use it on another. For example, in Figure 5 from the additive synthesis lecture, we see waveforms (in WaveEditors) generated from previous experiments still available for comparison as new waveforms are generated.

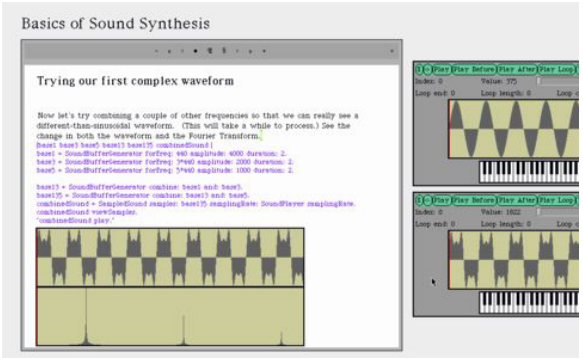

Figure 5: Wave Editors saving Prior Results in Experimentation

5 Conclusion

The computer music class described here is transferable to other departments with other curricula, because

the amount of music knowledge required is small and the programming level is well within the scope of CS1. We used Squeak because it runs on so many platforms (20+), offered many sound tools, an excellent programming infrastructure for exploring sound, and was known to our students.

The Active Essays we used are an innovative new medium that is well suited to this kind of domain. A text alone does not convey the domain of computer music well, especially for novices. An Active Essay provides the opportunity to see and hear experiments and to access tools for further experimentation within the lecture.

The course website is at `http://swiki.cc.gatech.edu:8080/compMusic` and the Active Essays themselves are available at `http://swiki.cc.gatech.edu:8080/compMusic/ActiveEssays`.

References

[1] AAUW. *Tech-Savvy: Educating Girls in the New Computer Age*. American Association of University Women Education Foundation, New York, 2000.

[2] Boulanger, R., Ed. *The CSound Book: Perspectives in Synthesis, Sound Design, Signal Processing, and Programming*. MIT Press, Cambridge, MA, 2000.

[3] Dodge, C., and Jerse, T. A. *Computer Music: Synthesis, Composition, and Performance*. Schimer:Thomason Learning Inc., 1997.

[4] Guzdial, M. *Squeak: Object-oriented design with Multimedia Applications*. Prentice-Hall, Englewood, NJ, 2001.

[5] Guzdial, M. Using squeak for teaching user interface software. In *Proceedings of ACM SIG Computer Science Education Conference*. ACM, Charlotte, NC, 2001, pp. 219–223.

[6] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. Back to the future: The story of squeak, a practical smalltalk written in itself. In *OOPSLA'97 Conference Proceedings*. ACM, Atlanta, GA, 1997, pp. 318–326.

[7] Roads, C. *The Computer Music Tutorial*. MIT Press, Cambridge, MA, 1996.